



Code less.
Create more.
Deploy everywhere.

Reducing the Cost of Defense and Aerospace Application Development with Qt


Abstract

Software developers in the defense industry necessarily consider a number of factors when selecting a toolset with which to construct applications. Chief among them are these five:

- Portability across processors and operating systems
- How much of the application the tool delivers directly off the shelf
- Richness, quality and number of the tool's developer productivity assets
- Real-time performance of the executable (especially if embedded)
- How much the application looks and feels "at home" in the target OS

The importance of these criteria explains why developers in this industry have increasingly turned to Qt, especially when applications call for a rich and powerful human interface.

Qt helps developers enable, build and deploy such systems in far less time with far fewer compromises in performance and portability while lowering costs both in development and when deployed.



Written by Randy Cronk
greatwriting.com

Table of Contents

- How Qt Helps Maximize Defense Industry Return on Software Investment3
- Critical Developer Requirements.....4
- Critical Application Requirements5
- Why Qt Succeeds in Defense6
- Where Qt Alternatives Fail8
- Unlock Return on Investment.....8
- Make Tradeoffs Obsolete9

How Qt Helps Maximize Defense Industry Return on Software Investment

Qt is a complete task-optimized developer's toolbox ideally suited for today's defense-related applications and their environments — both in development and when deployed.

Software developers in the defense industry necessarily consider a number of factors when selecting a toolset with which to construct applications for the defense industry. Chief among them are these five:

- Portability across processors and operating systems
- How much of the application the tool delivers directly off the shelf
- Richness, quality and number of the tool's developer productivity assets
- Real-time performance of the executable (especially if embedded)
- How much the application looks and feels "at home" in the target OS

The importance of these criteria explains why developers in this industry have increasingly turned to Qt, especially when applications call for a rich and powerful human interface. Examples of applications already developed and deployed using Qt include flight control, mission planning, tactical radio, robotics, and integrated battlefield sensing. In these and similar roles, agility depends on real-time situational awareness, communication, and response. Some of these issues are "classic UI," such as how to create an intuitive look and feel. Others go well beyond UI, such as how to integrate disparate data feeds from multiple sources as diverse as a database in Kansas, sensors in an urban war zone, and satellite surveillance.

Today's weapons are smaller, smarter and more tightly interconnected. Qt helps developers enable, build and deploy such systems in far less time with far fewer compromises in performance and portability while lowering costs both in development and when deployed.

At its core Qt is a cross-platform application framework — libraries of powerful C++ classes — specifically tailored to develop GUI programs and related functions. Rather than invent their own code, developers leverage Qt's off-the-shelf libraries in a task-optimized developer's "workshop" so they can more quickly build world-class applications that run more efficiently on all size processors, from ARM-9 to large Intel-based multi-processor systems. In addition to the code itself, Qt offers a number of highly polished productivity tools, including a drag-and-drop GUI builder, an optimized IDE, a help subsystem, and extensive internationalization support. Developers also benefit from best practices developed over decades of collective Qt experience across thousands of projects.

Another Qt benefit is that developers can stay within the same development tool longer — and not just for GUI development — increasing productivity and helping to ensure all parts of an application work well together. Qt's non-GUI features include SQL database access, XML parsing, thread management, WebKit-based HTML rendering, a multi-media subsystem, network support and a unified cross-platform API for file handling. It also includes packages such as data structures and a networking library. Platform dependency issues (e.g., I/O handling, bit order, address space) that foster bugs and slow down projects are also mitigated. This means that applications developed with Qt targeting one platform, can be moved to other platforms with a minimal amount of effort — typically just a re-compile.

The fact that Qt applications are compiled results in typically faster performance than the equivalent application written in Java, especially on embedded systems where processing and memory resources are often severely constrained and every last byte and CPU cycle must be conserved for the primary mission of the device. And since they are cross-platform they offer economies of scale and flexibility in the use of human and capital assets. Debugging applications on desktop systems is much faster than debugging on the target machine.

But perhaps the most important Qt feature is that all its features come in a single toolkit. Where other tool choices often pit one requirement against another, Qt lets developers satisfy most requirements with Qt alone. That applies both for what developers need in order to create applications and for how applications must perform when they are deployed. Once both sets of requirements are understood, it's easy to understand Qt's success in the defense space.

Critical Developer Requirements

Defense applications and their developers face critical challenges. They must ensure the survival of the nation and its war fighters in both asymmetric and symmetric battle-fields worldwide. So applications must be fast, maximally functional, easy to use, reliable, and perform over a lifetime of at least 20 years. This no-compromises policy easily multiplies initial development costs by two to three times that of typical commercial applications — with total lifecycle costs far exceeding this benchmark. And even as countries often have smaller defense budgets to spend, modern weapons get obsolete sooner unless upgraded — the effect of technology change that also benefits the enemy with cheaper and more lethal weapons.

Given these challenges, developers have clear requirements any would-be tool provider must satisfy — requirements that are fairly consistent across the space even if the requirements themselves may sometimes seem in opposition with each other.

Portability across processors and operating systems. Developers don't want to be locked into a platform just because that happens to be where the application was originally developed or deployed. Take the current migration of many workstation-based defense applications to PCs and Linux. Thanks largely to the computer games industry, there is now a rich supply of commodity hardware available offering very fast graphics. What developers want is the ability to mix and match this hardware based on features, performance and cost as competition and scale drive continued innovation and cost reductions. What they don't want is the legacy technology holding them back — i.e., the technology in which these applications were originally developed (e.g., Motif, X-Windows, Athena, OpenLook) — that makes the applications harder to port because available tools only run on, and only port to, UNIX workstations.

How much of the application the tool delivers directly off the shelf. Today's developers don't write code like their predecessors did — a style one retiring software engineer calls “algorithmic.” They expect to find reusable code they can, like hardware components, plug and play to configure a desired set of functions. Provided code may include C++ classes, class libraries, and higher-level abstractions whose APIs allow implementations coded in one language (e.g., C++) to be plugged into implementations coded in another (e.g., Ada.). The benefits of reusability are well documented: Developers don't waste time reinventing the wheel. Functions are easier to isolate and debug. Implementations reflect best practices — and so on.

Richness, quality and number of developer productivity assets. Beyond lots of pre-built functionality, developers can also look for other assets that speed deployment of critical functions to war fighters — assets like graphically oriented drag-and-drop GUI builders and IDEs (integrated development environments) with facilities like referenced code highlighting and help systems. Internationalization support is another example — such as having help in multiple languages and a layout engine that renders text based on a given language left-to-right or right-to-left and with proper line breaks. Yet a different kind of “asset” is community support — i.e., the pool of technology providers, consultants, independent developers, working groups and others who can provide expertise, coding examples, canned functions and other types of resources to developers.

Real-time application performance. Defense often means identifying and responding to threats quickly — as in air battles. Applications require fast hardware and optimized code — which typically calls for source code to be compiled rather than interpreted. The performance requirement (e.g., quick identification of an enemy) often opposes the portability requirement (e.g., quick identification from multiple types of aircraft). That’s because optimizations designed to exploit one hardware feature may be counter-productive on hardware that lacks the feature or implements it differently. So rather than implement the same source code the same way for each target, the tool should take these differences into account and optimize accordingly.

How much the application looks and feels “at home.” Optimizing the application for best hardware performance is one priority; optimizing it for best user performance is another. Users perform better if all the applications on their screens (e.g., vehicle control, sensors, and weapons control) all look and feel the same — i.e., with consistent use of colors and fonts; with consistent keyboard shortcuts; with controls that have consistent graphic representations, locations and behaviors; and so on. Not only should the applications be consistent; they should be consistent with the operating system that hosts them — so Linux apps look like Linux apps, and the same for Windows, Mac OS X or other OS.

Critical Application Requirements

If the critical requirement for applications can be stated in one word, it is agility — so war fighters can respond swiftly to new threats and new opportunities, so new capabilities come online quickly, so agencies are not left exposed to technology obsolescence, and so government benefits fully from the rapid innovation and cost reductions occurring in the commercial marketplace. Some specific examples:

A flight control system. A pilot views two displays — one a large animation of flight parameters, weapons status, and geo-coordinated positions of friend and enemy, both aircraft and missiles. A smaller touch screen nearby shows context sensitive commands for communications, weapons firing and flight control. Virtually everything the pilot needs to see or touch in order to survive and win in combat are on these screens — so powerful real-time GUI performance is critical. Success happens because applications developed in Qt can be compiled on today’s high performance graphics hardware. They’re also easily portable so the vendor can market its flight control system to many more customers, thereby leveraging one investment.

Satellite terminal configuration. Secure tactical communications to the battlefield requires swift application of preconfigured ground station parameters stored in a MySQL database and controlled via a Qt-enabled interface. Previously implemented in Microsoft Access 97, a straightforward migration to Qt, Linux, and 21st century hardware dramatically improved response time, as did Qt’s ready-made MySQL integration and a Windows-like interface users already knew.

A mission planning system. Flight training maneuvers involving various battle scenarios are fully choreographed on animated displays and discussed prior to take off. After pilots return, the actual mission is downloaded from the aircrafts — allowing pilots and planners to “re-fly” both missions — planned and actual — and compare them side-by-side in 3-D animation. This includes all sensor and flight control data, as well as visual representations of friend and enemy — relative positions, flight vectors, speeds, weapons firings, “kills” and misses. Qt’s highly functional widgets, compiled runtimes, and support for commercially available high-end PC graphics cards are key.

An equipment maintenance system. For over a decade, this system has tracked every maintenance detail for thousands of pieces of equipment representing millions of dollars in inventory — its location, condition, what was done to it, when it was done, what parts were involved, who performed the work, and so forth. However, as new equipment came into inventory and maintenance histories grew longer, the maintenance system itself became difficult to maintain. Based on a forms-type product its vendor no longer supported, the legacy workstation-based system lacked flexibility, scalability, extensibility and a PC-style interface that users could easily operate. So a complete re-implementation on PC hardware was ordered — creating a significant upgrade in usability (from Qt’s visually intuitive widgets) and speed (from Qt’s compiled runtime leveraging faster hardware).

A Warfighter Machine Interface (WMI). Soldiers in tanks and other ground vehicles receive information, control equipment and communicate with one another using this one interface. According to military commanders, the WMI is “where a soldier is going to spend 99% of his time. It’s the first thing soldiers see — the first thing they use — and the first thing they need to understand.” In other words, combat agility often means ease-of-use. “Soldiers need to know ‘How do I make this work? How do I talk to my commander? How do I issue orders?’” Here that results from a consistent look-and-feel that’s also highly intuitive and married to high functionality — again, by leveraging Qt.

What’s clear from these applications is that they are (and must be):

- **Fast** — so people and systems can respond rapidly to threats (hence, the “real-time executable performance” requirement cited earlier)
- **Memory efficient** — to accommodate battlefield deployments (e.g., inside aircraft, ground vehicles, infantry packs, etc.)
- **Visually intuitive** — to rapidly integrate onscreen multiple information sources in a meaningful way and correctly interpret them
- **Easy to control** — meaning screen objects not only make sense but respond efficiently to user actions
- **Easy to interconnect** — because before applications can present information in a timely way, they must first receive it and that often means from multiple and disparate sources
- **Easy to change** — since military situations and needs continually evolve, sometimes rapidly, so will applications need to in their functions, features and external interfaces

Selecting an appropriate developer toolset means mapping developers’ requirements against these application requirements. You can also compare toolsets on those benchmarks versus a prevailing defense industry standard — Qt.

Why Qt Succeeds in Defense

Qt’s success in defense is no accident — in view of its history and the many features it provides to support both agile applications and their developers.

Defense applications tend to be long lived and so is Qt. It’s long been the de facto standard C++ framework — i.e., libraries of pre-built application classes — for high performance cross-platform software development. Qt was developed starting in 1991 by the founders of Trolltech. Nokia acquired Trolltech in 2008 and changed the name to Qt Development Frameworks, which offers Qt under both commercial and open source licenses. (For an in-depth comparison of available licenses, see <http://qt.nokia.com/products/licensing>.) Since 1995, Qt has been widely used by organizations such as Adobe®, HP®, Google®, the European Space Agency, Thomson Reuters®, Texas Instruments® and Skype®. In addition to Qt’s role as an application development framework, the Open Source edition of Qt is the foundation of KDE, the Linux desktop environment.

Today Qt is much more than a framework as it comprises a fully featured toolset that includes the C++ class libraries plus an extensive array of tools supporting “write once, compile anywhere” development. Cross-platform development is fast and straightforward, especially for highly agile defense applications — i.e., *those that must be very memory efficient and/or satisfy significant GUI, data, communication, and runtime performance demands.*

Using a single source tree and a simple recompilation, Qt applications can be written for Windows, Mac OS X, Linux, Solaris, HP-UX, and many other versions of UNIX with X11. They can also be compiled to run on embedded platforms Linux, Symbian and Windows CE — and embedded real-time platforms QNX and VxWorks. Qt has excellent cross-platform support for multimedia and 3D graphics, internationalization, SQL, XML and unit testing. It also provides platform-specific extensions for specialized applications and to provide an “at home” look and feel consistent with the target platform. It is this ability to exploit underlying platform-specific features that enables the many defense industry developers who prefer Qt even for single-platform development on Windows, Mac OS X, and UNIX. Should the need ever arise, however, they can also easily port the same code base to different platforms — ranging potentially from embedded ARM solutions to x86 blade servers.

For applications that must render or edit web content, Qt provides an out-of-the-box web browser engine called QtWebKit, based on the Open Source WebKit. Integration with Qt networking classes enables web pages to be transparently loaded from web servers, the local file system or even the Qt resource system. This engine also provides a bridge between the JavaScript execution environment and the Qt object model, so native application code can be scripted. Qt’s ability to integrate rich client and web content greatly increases the speed at which defense users can access critical information. One example based on Google Maps displays latitude and longitude of any point on a map just by touching it.

Other Qt advantages for defense include the extensive out-of-the-box classes for sending, receiving, handling and storing data in efficient and standard-compliant way. These include SAX and DOM classes for reading and manipulating data stored in XML-based formats. Objects can persist in memory using Qt’s STL (Standard Template Library) compatible collection classes, and handled using styles of iterators used in Java® and the C++ STL. Qt’s input/output and networking classes provide local and remote file handling using standard protocols. Such pre-fab application plumbing also includes a TCP socket wrapper for built-in device interfacing.

Another example — one that also supports defense applications’ robust connectivity needs — is signals and slots. These facilities connect application components together so they can communicate in a simple type-safe way — replacing the old and insecure callback technique used in many legacy frameworks. Qt also provides a conventional event model for handling mouse clicks, key presses and other user input.

For data-intensive applications that can also be platform-independent, Qt includes native drivers for Oracle®, Microsoft® SQL Server, Sybase® Adaptive Server, IBM DB2®, PostgreSQL™, MySQL®, Borland® Interbase, SQLite, and ODBC-compliant databases. Qt also includes database-specific widgets, i.e., prebuilt onscreen controls, and, in addition, developers can easily make any built-in or custom widget data-aware.

Outside the class libraries themselves, Qt offers many productivity assets to help defense industry developers meet tight deadlines faster with less rework. Those include an IDE (Qt Creator), help system (Qt Assistant), and a powerful graphical GUI builder (Qt Designer) that is not limited just to building GUIs. It is also effective for creating entire applications and has integrated support for many popular IDEs. Qt Designer also supports Qt’s powerful layout features in addition to absolute positioning. Qt also offers excellent support for multimedia and 3D graphics. Qt’s painting system offers high quality rendering across all supported platforms and allows experts to further tune low level graphics performance to take advantage of any hardware acceleration offered

by the embedded hardware. A sophisticated canvas framework (GraphicsView) enables developers to create interactive graphical applications that take advantage of Qt's many advanced painting features. Finally, of course, Qt also supports all the user interface functionality required by modern applications, such as menus, context menus, drag and drop, and dockable toolbars.

Having all these features available right out of the box, platform independent, and supported by a host of advance productivity tools gives agencies a huge edge in building today's defense applications. That edge is clearly seen when compared to alternative approaches such as Java or platform-specific tools, such as the SDKs from OS suppliers.

Where Qt Alternatives Fail

The key advantage over Java is obvious: compiled code runs faster than interpreted code and often can be more finely tuned for enhanced functionality. This advantage over Java is especially great in embedded systems where every last byte and CPU cycle needs to be dedicated to the military objective of the device. Another alternative is the homegrown approach — meaning that developers isolate functionality as discrete modules that they deploy on top of a custom-built portability layer. The premise is that they can more quickly write and port applications across platforms by plugging functions into the layer, rather than by writing them new for each application or for each platform. Where the premise fails is that the portability layer itself becomes an extra development and maintenance burden for developers. And it's one typically outside their core competency. Resources that would otherwise have been focused on the application are instead focused on making sure the portability layer "works" regardless of changing platforms, operating systems, hardware specs and other factors — perhaps for years.

The disadvantages of platform-specific SDKs are equally clear. Differences such as bit order (big endian or little endian), address space (32-bit or 64-bit), and I/O handling (asynchronous or synchronous) create problems when porting apps. Rewriting code to resolve these differences slows down development, diverts resources from other priorities and creates more opportunities for bugs. Multiple code versions cost more and take longer to develop, maintain and upgrade. Platform-specific optimizations take longer and require more specialized and costly expertise.

In contrast, Qt has been in use for over 15 years, and has been hardened by having to support thousands of applications of every type. And if an application has some unique need, the delivery of the source code to Qt with all license types – commercial and open source – make it possible for developers to customize it.

Unlock Return on Investment

All these alternatives limit return on investment in developing agile defense-oriented applications. Specialized resources that can't be employed on multiple platforms are harder to leverage. Developers and tools may sit idle, and products may find fewer customers. For contractors, that reduces profits and limits growth. For agencies, that increases costs, reduces functionality, and lowers system performance — none of which helps in a time of budget cuts and accelerating technology change.

Compare this to Qt — which is to have a unified development team, a comprehensive set of reusable classes, a rich set of productivity tools — all while still retaining the ability to optimize a single code base against multiple targets of opportunity economically and swiftly.

Make Tradeoffs Obsolete

The bottom line for both agencies and contractors is that tradeoffs once considered unavoidable, are made obsolete. Whether to select one platform for its speed and another for its enhanced look and feel is now a false choice. So is having to choose between the platform with the largest installed base or the one with the most advanced features — or having to decide between a generic “one-size-fits-all” implementation or one cleanly optimized for the target environment.

When it comes to the critical overarching priorities — like faster time to market, lower cost, reusable assets, more finely tuned apps, better user experience, and higher performance — development teams may wish to consider what they are giving up if not choosing Qt. Today’s war fighting applications must deliver maximum agility — and their developers demand it.

About Qt

Qt is a cross-platform application framework. Using Qt, you can develop applications and user interfaces once, and deploy them across many desktop and embedded operating systems without rewriting the source code. Qt Development Frameworks, formerly Trolltech, was acquired by Nokia in June 2008. For more details about Qt please visit <http://qt.nokia.com>.

About Nokia

Nokia is the world leader in mobility, driving the transformation and growth of the converging Internet and communications industries. We make a wide range of mobile devices with services and software that enable people to experience music, navigation, video, television, imaging, games, business mobility and more. Developing and growing our offering of consumer Internet services, as well as our enterprise solutions and software, is a key area of focus. We also provide equipment, solutions and services for communications networks through Nokia Siemens Networks.



**Code less.
Create more.
Deploy everywhere.**

NOKIA